Pointer Analysis

CMPUT 416/500
Foundations of Program Analysis

Karim Ali
@karimhamdanali

- Call graph = calling relationships at compile time

- Sound (no missing edges)

- Precise (few spurious edges)

- On-the-fly call graph construction

Previously

- Rapid Type Analysis (RTA)

  - single points-to set for the program

- Variable Type Analysis (VTA)

  - field-based, simplify SCC, no OTF

- Andersen-style analysis (SPARK)

  - separate points-to sets for each variable

- Steensgaard-style

  - equality-based (not subset-based)

Today

other variants of points-to

Points-to Analysis
vs
Alias Analysis

Points-to Analysis

Points-to Analysis

v

Points-to Analysis

$$\text{points-to}(v) = \{o_1, o_2, \ldots \}$$

Alias Analysis

Alias Analysis

$$v_1 \quad v_2$$

Alias Analysis

$$alias(v_1, v_2) = true/false$$

Points-to vs Alias

allocation sites

$$\text{points-to(v)} = \{a_1, a_2, ... \}$$

may/must

Points-to vs Alias

$$\text{points-to}(v) = \{a_1, a_2, \ldots\}$$

$$\text{may-alias}(v_1, v_2) = \text{true/false}$$
$$\text{must-alias}(v_1, v_2) = \text{true/false}$$

May-Alias vs Must-Alias

```
a = new A();
if(..) {
  b = a;
}
c = new C();
d = c;
```

may-alias(a,b) =   true

must-alias(a,b) = false

may-alias(a,c) = false

must-alias(c,d) =   true

May-Alias vs Must-Alias

```
a = new A();
if(..) {
  b = a;
}
c = new C();
d = c;
```

may-alias(a,b) =   true

must-alias(a,b) =   false

may-alias(a,c) =   false

must-alias(c,d) =   true

Must-alias is typically associated with control flow!

Must-Alias => Flow-Sensitive?

```
    b = null;
    d = null;
s₁: a = new A();
    if(..) {
      b = a;
    }
s₂: c = new C();
    b = c;
s₃: d = a;
```

$\text{must-alias}(a,s_1,d,s_2) = \text{false}$

$\text{must-alias}(a,s_1,d,s_3) = \text{true}$

$\text{must-alias}(b,s_2,c,s_1) = \text{false}$

$\text{must-alias}(b,s_2,c,s_3) = \text{false}$

after s_1

after s_2

Must-Alias => Flow-Sensitive?

```
       b = null;      must-alias(a,d) =        false
       d = null;      must-alias(a,d) =        true
s₁: a = new A();
       if(..) {
                      must-alias(b,c) =        false
         b = a;       must-alias(b,c) =        false
       }
s₂: c = new C();
       b = c;
s₃: d = a;
```

Have to be conservative!

Must-Alias => Flow-Sensitive?

```
      b = null;
      d = null;
s₁:  a = new A();
      if(..) {
        b = a;        must-alias(a,d) =      false
      }               must-alias(b,c) =      false
s₂:  c = new C();
      b = c;
s₃:  d = a;
```
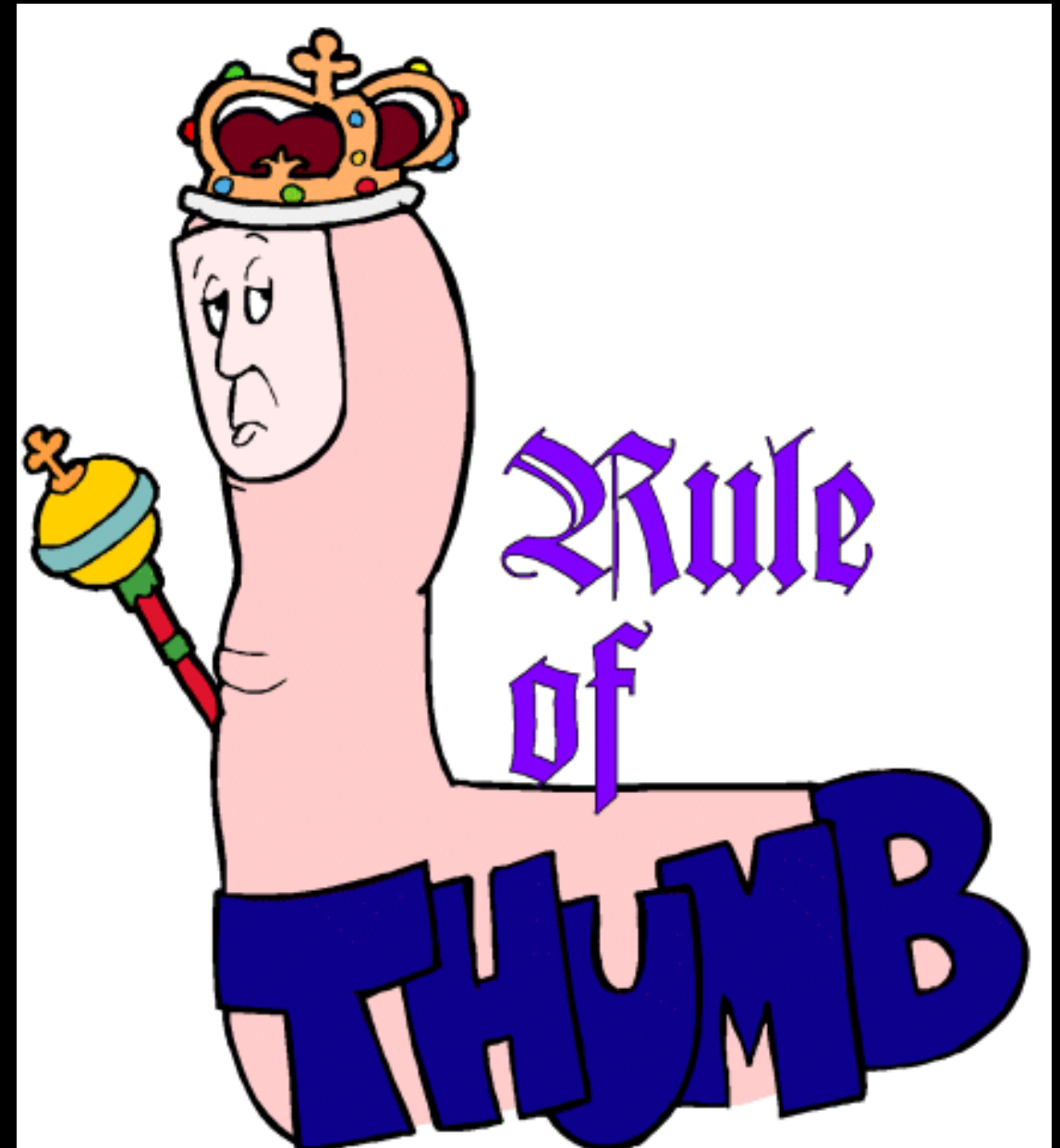
- Must-X analyses must be flow-sensitive.
- May-X analyses may be flow-(in)sensitive.

Points-to as Alias

$$\text{alias}(v_1, v_2) =$$
$$\text{points-to}(v_1) \cap \text{points-to}(v_2) \neq \emptyset$$

When to use Alias Analysis?

Using Alias Analysis

```
void readProp(String id, String d) {
    String s = Properties.read(id);
    if(s==null) s = d;
    return s;
}
```

Assume you can't analyze Properties.read()
(e.g., native method, unknown library)

Using Alias Analysis

```
void readProp(String id, String d) {
    String s = Properties.read(id);
    if(s==null) s = d;
    return s;
}

may-alias(s, d) = true
```

Using Alias Analysis

```
void readProp(String id, String d) {
    String s = Properties.read(id);
    if(s==null) s = d;
    return s;
}
```

may-alias(s, d) = true

points-to(s) = ∅   unsound

points-to(s) = any object   imprecise

Using Alias Analysis

```
void readProp(String id, String d) {
  String s = Properties.read(id);
  if(s==null) s = d;
  return s;
}
```

may-alias(s, d) = true

points-to(s) = ?

points-to(d) = ?

may-alias(s, d) =
  points-to(s) ∩ points-to(d) ≠ ∅

- Associating variables with allocation sites is either unsound or imprecise (i.e., points-to)

- Alias analysis is better suited, because it can reason about the relationship between variables without caring about which objects they point to

"Direct" Alias Analysis

"Direct" Alias Analysis

```
b = a;
```

```
c = b;
```

$$\text{may-alias}(b,a) = \text{true}$$

$$\text{may-alias}(c,b) = \text{true}$$

"Direct" Alias Analysis

```
a = null;
```
$$\text{may-alias}(b,a) = \text{true}$$
```
b = a;
```
$$\text{may-alias}(c,b) = \text{true}$$
```
c = b;
```

usually ignored!

$$\text{may-alias}(v_1, v_2) = \text{may-alias-or-both-null}(v_1, v_2)$$

When to use Points-to Analysis?

Using Points-to Analysis

$$a_1: \text{l = new LinkedList();}$$
$$\text{l.clear();}$$

points-to(l) = { a_1 }

type-of(points-to(l)) = { LinkedList }

l.clear() can only invoke LinkedList.clear()

For method de-virtualization,
alias analysis has almost no use

Weak Updates
vs
Strong Updates

Weak Updates

- Doable if only `may-alias` info is available

- Retain previous info, and add to it

- Cannot *kill* old info (leads to unsound results)

Weak Updates

- constant propagation
- variables initialized to 0
- only `may-alias(x,y)` is known

$x.f \mapsto 0 \quad y.f \mapsto 0$

`x.f = 3;`

$x.f \mapsto 3 \quad y.f \mapsto 3$

must retain old values of `x.f` and `y.f`

$x.f \mapsto 0 \quad y.f \mapsto 0$

Strong Updates

- constant propagation
- variables initialized to 0
- `must-alias(x,y)` is known

$$x.f \mapsto 0 \quad y.f \mapsto 0$$

$$x.f = 3;$$

$$x.f \mapsto 3 \quad y.f \mapsto 3$$

can kill old values of
`x.f` and `y.f`

$$x.f \mapsto 0 \quad y.f \mapsto 0$$

Access Paths

local variable

v . f . g . h . . .

field accesses

Access Paths as Object Descriptors

```
x = new X();

a.f = x;

b.g = a.f;

c.h = b;
```

```
{ x }

{ x, a.f }

{ x, a.f, b.g }

{ x, a.f, b.g, c.h.g }
```

$$\{x\}, \{x,y\}$$

Encoding Alias Info as Access Paths

```
x = new ..();
y = new ..();
z = new ..();
```

{x}, {y}, {z}

`if(..)`

{x}, {y}, {z}   {x}, {y}, {z}

`y = x;`

{x}, {y}, {z}, {x,y}   {x,y}, {y}, {z}

`z = x;`

{x,z}, {y}, {z}, {x,y,z}

Encoding Alias Info as Access Paths

- **may-alias(x,y)** if there is a set containing both **x** and **y**

- **must-alias(x,y)** if each set that contains **x** also contains **y**

```
may-alias(x,y)  = true
must-alias(x,z) = true        {x,z}, {x,y,z}
must-alias(x,y) = false
```

Strong Updates with Access Paths

Strong Updates with Access Paths

```
x = new ..();
y = new ..();
z = new ..();
```

{x}, {y}, {z}

if(..)

{x}, {y}, {z}   {x}, {y}, {z}

```
y = x;
```

{x,y}, {z}

{x}, {y}, {z}, {x,y}

```
z = x;
```

{x,z}, {y}, {x,y,z}

Strong Updates with Access Paths

- constant propagation
- variables initialized to 0

{x}, {y}, {z}, {x,y}

$\{x\}.f \mapsto 0, \{y\}.f \mapsto 0,$
$\{z\}.f \mapsto 0, \{x,y\}.f \mapsto 0$

z = x;

{x,z}, {y}, {x,y,z}

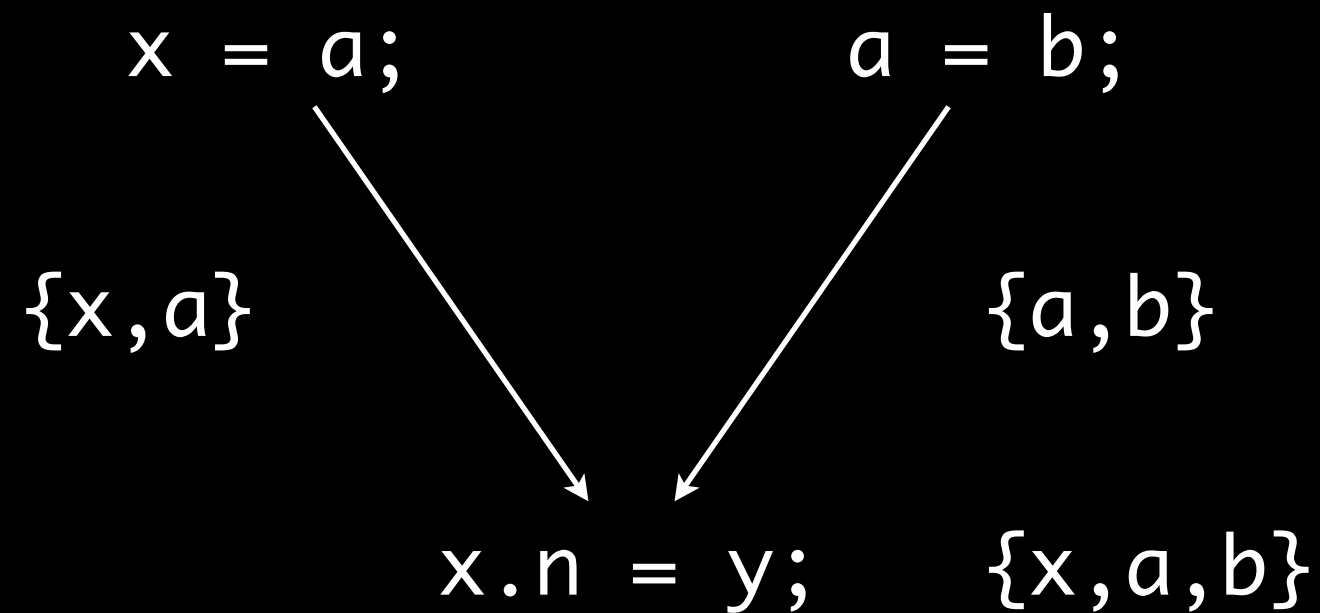$\{x,z\}.f \mapsto 0, \{y\}.f \mapsto 0,$
$\{x,y,z\}.f \mapsto 0$

x.f = 3;

Strong Update

{x,z}, {y}, {x,y,z}

Pointer Analysis
&
Distributivity

Pointer Analysis & Distributivity

$$x = a; \qquad a = b;$$

$\{x,a\}$ $\qquad\qquad\qquad$ $\{a,b\}$

$$x.n = y; \qquad \{x,a,b\}$$

imprecise but distributive

Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural Pointer Alias Analysis. ACM Transactions on Programming Languages and Systems 21, 4 (July 1999), 848-894.

- In general, precise pointer analysis is **not** distributive

- Merging first yields different results than merging later

- f(x ⊔ y) ≠ f(x) ⊔ f(y)

Summary

- Certain Points-to analyses can be used to also answer alias-analysis queries

  - Advantage: re-use points-to analysis results

- Must-alias => flow-sensitive setting

- Strong update requires must-alias information

- Flow-sensitive points-to analysis is not distributive

Next

- Inter-Procedural Analysis